INTERPRETER DESIGN PATTERN NOTES

BY: TEAM TRES (Canonigo, Garcia, Tabañag)

I. BRIEF HISTORY

The Interpreter pattern was first introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (commonly known as the Gang of Four or GoF). This influential book, published in 1994, formalized the concept of design patterns and provided a template for solving common software design problems.

II. PROBLEM

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

III. DEFINITION

Interpreter design pattern falls under the category of **behavioral design patterns**. It is used to **define a grammatical representation for a language** and provides an interpreter to deal with this grammar. This pattern involves implementing an expression interface which tells you to interpret a particular context.

IV. EXAMPLES

1. **Programming Language Compilers/Interpreters**
   The Interpreter pattern can be used to parse and evaluate the syntax and semantics of the language.
2. **Regular Expressions**
   Regular expressions define a grammar for specifying text patterns, and the interpreter is responsible for matching and extracting text based on those patterns.
3. **Database Query Language**
   The SQL query is interpreted to retrieve data from the database.

V. KEY COMPONENTS

- **Client** - builds (or is provided) the AST assembled from TerminalExpression and NonTerminalExpression.
- **Context** - contains information that's global to the interpreter.
- **Abstract Expression** - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **Terminal Expression** - implements an Interpret operation associated with terminal symbols in the grammar.
- **NonTerminal Expression** - implements an Interpret operation for nonterminal symbols in the grammar.

VI. ADVANTAGES

- Flexibility for Domain-Specific Languages
- Ease of Grammar Changes
- Separation of Concerns
- Customization

VII. DISADVANTAGES

- Complexity for Complex Grammars
- Performance Overhead
- Verbose Implementations

VIII. WHEN TO USE THE INTERPRETER DESIGN PATTERN

1. **There is no complex grammar involved.**
   - For complex grammar, the interpreter can take extensive space and time, leading to an unimaginable class hierarchy for grammar.
2. **When efficiency is not the focus**
   - Most efficient translators first convert the regular expressions into another form before translating. So, use them when your focus is reusability and easy, not efficiency.
3. **When expressions can be represented in Syntax Trees**
   - An interpreter design pattern utilizes abstract syntax trees to read and decode language.